#1 JDK

**JDK (Java Development Kit)**: The JDK is used by developers to write, compile, and debug Java code.

JDK = JRE + development tools like javac

#JRE: **JRE = JVM + Libraries**

JRE stands for Java Runtime Environment.

JRE contains all the built-in classes, libraries, and other files needed to run Java programs.

JRE does not validate bytecode; instead, the JVM performs bytecode verification.

JRE provides the runtime environment and includes the necessary libraries and JVM, which loads and executes classes.

#2

JVM stands for Java Virtual Machine.

JVM is the environment where Java bytecode is executed.

JVM makes Java platform-independent by executing bytecode on any OS(like windows , like ) with a compatible JVM.

JVM is platform-dependent, but Java is platform-independent **because of JVM**.

JVM is platform-dependent because each OS needs its own implementation of JVM.

You compile Java code using JDK (Java Development Kit), which produces bytecode (.class file). JVM Converts **bytecode into machine code** specific to the system (Windows, Linux, etc.).

Tool Role

javac Compiles Java code into bytecode

java Runs the compiled bytecode using JVM

**The main() method in the java :**

the main() method in Java has a fixed signature:

public static void main(String[] args)

1. public

It's an access modifier.

It means JVM can access it from anywhere — even outside the class.

If main is not public, JVM can’t run it, and you’ll get a runtime error.

🗣 Think of it as “main() needs to be visible to the JVM.”

2. static

main() is static so JVM can call it without creating an object of the class.

This saves memory and avoids unnecessary object creation.

If it's not static, the JVM won’t know how to call it (since there’s no object yet), and you’ll get an error.

🗣 It allows the JVM to start execution directly.

3. void

This means the method does not return anything.

After main() runs, the program ends, so there's no need to return any value to the JVM.

🗣 JVM doesn’t expect any result from main().

4. main

This is the name of the method that the JVM looks for.

It’s not a keyword, but it is fixed — you cannot change it.

🗣 Changing the name will stop the program from starting.

5. String[] args

This is used to accept command-line arguments.

It's an array of Strings.

You can change the variable name (like String[] data is also valid), but type must be String array.

Example:

class Demo {

public static void main(String[] args) {

for (String arg : args) {

System.out.println(arg);

}

}

}

Run like:

java Demo hello world

Output:

hello

world

✅ Alternate Forms (Still Valid):

These are also accepted by JVM:

public static void main(String args[])

public static void main(String... args)

✅ Final Line:

In short, main() is the starting point of Java execution, and every part of its signature is essential for the JVM to find and execute it properly.

**Method :**

A **method** in Java is a block of code that performs a specific task. It is used to **reuse code**, improve **readability**, and **organize logic**.

Java Memory Model in JVM (Stack vs Heap)

**1. What is Stack Memory in Java?**

* Used for **method execution** and **local variables**.
* Every method will have its own stack
* Every time a method is called, a **stack frame** is created.
* Stores:
  + Local variables
  + Method parameters
  + Reference variables (which point to objects in heap)
* Stack frames are removed **after method ends**.
* Memory is **automatically managed** (LIFO: Last In, First Out).

**What is Heap Memory in Java?**

* Used to store **objects** and their **instance variables**.
* Objects created using new are stored here.
* Shared across all threads.
* **Garbage Collector (GC)** automatically frees unused heap memory.

3. Scope of Instance and Local Variable :

Instance Variable => inside class ,outside method , instance variable are the part of the object

Local Variable=> Inside the method only

5. Object Creation and Memory Flow

Calculator obj1= new Calculator();

Obj1(reference variable) is stored inside the stack

Calculator obj and its variables is stored inside the Heap

3. Method Area

Stores:

o Class-level metadata

o Static variables (none in this program)

o Method bytecode

Shared among all threads

🔸 1. Instance variables are stored in the heap

When you create an object using new, it is stored in heap memory.

All instance variables (fields inside the class) are part of that object.

So, instance variables live in the heap.

new Calculater() creates an object in the heap

🔸 2. Local variables are stored in the stack

When a method is called, a stack frame is created.

All local variables and parameters are stored in this stack.

The stack is removed after method execution ends.

It disappears when the method finishes.

🔸 4. Reference variables are in the stack but point to heap

A reference variable (like obj) is created in the stack.

It holds the memory address of the object created in the heap.

Calculator obj = new Calculator();

obj is in **stack**, pointing to an object in **heap**.

//String

A String is a non-primitive data type that represents a sequence of characters.

Java provides two ways to store character sequences:

Character array: Requires manual handling : array (char[]) instead of a String

String object: Simplifies working with character sequences.

String str = "shiva"; // String object

2. How to Create a String Object in Java?

Strings in Java can be created in two ways:

1️⃣ Using the new Keyword (Heap Memory)

String str = new String("shiva");

This creates a new object in Heap Memory, even if "shiva" already exists in the String Pool.

It does not reuse the existing object from the String Pool.

Less memory efficient because it forces Java to allocate new space.

2️⃣ Using String Literal (Stored in String Pool)

String str = "shiva";

Stored in the String Pool, which is part of the Heap.

If "shiva" already exists in the String Pool, Java reuses it instead of creating a new object.

More memory efficient because Java avoids duplication.

The String Pool (also called the String Intern Pool) is a special area inside the Heap Memory where Java stores string literals

A literal in Java is a fixed constant value that is directly written in the code. It represents a specific value that does not change during execution.

String str1 = "shiva"; // Stored in String Pool

String str2 = "shiva"; // Reused from String Pool

String str3 = new String("shiva"); // Created in Heap (new object)

System.out.println(str1 == str2); // ✅ true (Same reference in String Pool)

System.out.println(str1 == str3); // ❌ false (Different objects in memory)

👉 Note:

== checks reference equality (whether they point to the same object).

equals() checks content equality.

3. What is a HashCode?

HashCode is a unique number assigned to objects in the Heap Memory.

It helps in identifying objects and is used in hash-based collections (e.g., HashMap, HashSet).

Every String object has a unique hashcode, unless it is explicitly overridden.

Example:

String str1 = "shiva";

String str2 = new String("shiva");

System.out.println(str1.hashCode()); // Same hashcode

System.out.println(str2.hashCode()); // Same hashcode but different memory reference

Even though both str1 and str2 contain "shiva", they refer to different objects if new is used.

4. String Concatenation (+ Operator)

The + operator is used to concatenate (join) strings.

Example:

String firstName = "shiva";

String lastName = "srivastava";

String fullName = firstName + lastName;

System.out.println(fullName); // Output: shivasrivastava

🔹 Note: String concatenation using + internally uses StringBuilder, which improves performance.

5. Commonly Used String Methods

✅ Finding String Length

System.out.println(str.length()); // Output: 5 (for "shiva")

✅ Getting a Character at a Specific Index

System.out.println(str.charAt(0)); // Output: 's'

In arrays, length is a property (arr.length).

In Strings, length() is a method (str.length()).

///

1. StringBuffer and StringBuilder in Java

StringBuffer and StringBuilder are mutable classes used to create and modify strings efficiently.

Unlike String, which is immutable, both allow modifications without creating a new object.

2. Important Methods of StringBuffer

Creating a StringBuffer object

StringBuffer sb = new StringBuffer("Hello");

Finding capacity of StringBuffer

System.out.println(sb.capacity()); // Default capacity is 16 + length of the string

Finding the length of the created StringBuffer

System.out.println(sb.length());

Appending content to an existing StringBuffer

sb.append(" World");

Converting StringBuffer to a String

String str = sb.toString();

System.out.println(str);

Deleting a character at a specific index

sb.deleteCharAt(3); // Deletes character at index 3

Inserting text at a specific index

sb.insert(0, "Hi "); // Inserts "Hi " at index 0

System.out.println(sb);

3. Difference Between StringBuffer and StringBuilder

Feature StringBuffer StringBuilder

Thread Safety Thread-safe (Synchronized) Not thread-safe (Faster)

Performance Slower due to synchronization Faster due to no synchronization

Use Case Suitable for multi-threaded applications Suitable for single-threaded applications

Yes! Your explanation is clear and covers key concepts. However, if asked about thread safety, you can briefly mention:

"Thread safety means multiple threads can access the same object without causing inconsistent results. StringBuffer achieves this by using synchronized methods, whereas StringBuilder does not synchronize methods, making it faster but not safe for concurrent use."

Object:

Java is an Object-Oriented Programming (OOP) language.

In the real world, everything is an object (e.g., pen, mouse, glass).

Every object has two main aspects:

Properties (Attributes) – What the object knows (e.g., a pen has color, brand, ink type).

Behavior (Methods) – What the object does (e.g., a pen can write, click, or uncap).

An object can have multiple properties and multiple behaviors.

To create an object, we first need a class.

A class acts as a blueprint for objects. It defines the properties and behaviors that objects of that class will have.

#2

In Java, objects are created using the new keyword or other object-creation techniques (e.g., reflection, cloning, deserialization).

The JVM is responsible for managing memory and object allocation, but it does not create objects automatically.

Before an object can be created, a class (blueprint) must exist.

The process works as follows:

You write Java code and define a class (blueprint).

The Java compiler (javac) compiles the class into bytecode (.class file).

The JVM loads the bytecode and executes it.

When an object is created using new, JVM allocates memory for it on the heap.

class :

In Java, everything is done inside a class.

A class is a user-defined blueprint or prototype from which objects are created.

A class contains methods and variables.

A class can be defined in a Java file, and multiple classes can exist within the same file.

A class itself does not occupy memory until an object is created.

Syntax of a class:-

class classname

{

methods, variables

}

#2

Object and Its Properties:

Every object has two main characteristics: properties (variables/attributes) and methods (functions/behaviors).

When an object of a class is created, the class is said to be instantiated.

All instances (objects) share the attributes and behavior of the class.

Methods and Access Modifiers:

Actions in a class are performed using methods.

We need to specify the access modifier of a method.

If a method should be accessible from anywhere, we use public.

If a method returns a value, we must specify its return type (e.g., int for integers).

Java is a statically typed language, meaning that the return type must be explicitly mentioned.

#3

Calling a Method from a Different Class:

To call a method from another class, we need to create an object of that class.

Creating an Object:

ClassName referenceVariable = new ClassName();

Here, referenceVariable is used to establish a reference to an object of the class.

The new keyword is used to allocate memory for the object.

Calling a Method Using an Object:

A method can be called using the reference variable of the object:

referenceVariable.methodName();

If a method requires arguments, we need to pass values while calling it:

referenceVariable.methodName(value1, value2);

The method must also define parameters to accept the passed values, ensuring the data type matches.

constructor :

In Java, It is a special type of method that is used to initialize the object

.It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. Every time an object is created using the new() keyword, at least one constructor is called. Rules for writing constructors are as follows:

The constructor(s) of a class must have the same name as the class name in which it resides.

A constructor in Java can not be abstract, final, static, or Synchronized.

Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

Types of Constructors in Java

1]Default Constructor

2]Parameterized Constructor

3]Copy Constructor

1. Default Constructor in Java

A constructor that has no parameters is known as default constructor. A default constructor is invisible. And if we write a constructor with no arguments, the compiler does not create a default constructor. It is taken out. It is being overloaded and called a parameterized constructor. The default constructor changed into the parameterized constructor. But Parameterized constructor can’t change the default constructor. The default constructor can be implicit or explicit.

Implicit Default Constructor: If no constructor is defined in a class, the Java compiler automatically provides a default constructor. This constructor doesn’t take any parameters and initializes the object with default values, such as 0 for numbers, null for objects.

Explicit Default Constructor: If we define a constructor that takes no parameters, it’s called an explicit default constructor. This constructor replaces the one the compiler would normally create automatically. Once you define any constructor (with or without parameters), the compiler no longer provides the default constructor for you.

2. Parameterized Constructor in Java

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

Remember: Does constructor return any value?

There are no “return value” statements in the constructor, but the constructor returns the current class instance. We can write ‘return’ inside a constructor.

3. Copy Constructor in Java

Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object.

Note: In Java,there is no such inbuilt copy constructor available like in other programming languages such as C++, instead we can create our own copy constructor by passing the object of the same class to the other instance(object) of the class.

Copy Constructor code:

// Java Program for Copy Constructor

import java.io.\*;

class Geek {

// data members of the class.

String name;

int id;

// Parameterized Constructor

Geek(String name, int id)

{

this.name = name;

this.id = id;

}

// Copy Constructor

Geek(Geek obj2)

{

this.name = obj2.name;

this.id = obj2.id;

}

}

class GFG {

public static void main(String[] args)

{

// This would invoke the parameterized constructor.

System.out.println("First Object");

Geek geek1 = new Geek("Avinash", 68);

System.out.println("GeekName :" + geek1.name

+ " and GeekId :" + geek1.id);

System.out.println();

// This would invoke the copy constructor.

Geek geek2 = new Geek(geek1);

System.out.println(

"Copy Constructor used Second Object");

System.out.println("GeekName :" + geek2.name

+ " and GeekId :" + geek2.id);

}

}

//Abstraction :

Abstraction in Java is the process of hiding the implementation details and only showing the essential functionality or features to the user. This helps simplify the system by focusing on what an object does rather than how it does it.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerator will increase the speed of a car or applying brakes will stop the car, but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc. in the car. This is what abstraction is.

Abstract Classes and Abstract Methods

An abstract class is a class that is declared with an abstract keyword.

An abstract method is a method that is declared without implementation.

An abstract class may or may not have all abstract methods. Some of them can be concrete methods

A abstract method must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.

Any class that contains one or more abstract methods must also be declared with an abstract keyword.

There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator.

An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

//Encapsulation:

Encapsulation in Java is a fundamental OOP (object-oriented programming) principle that combines data and methods in a class.

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates.

Another way to think about encapsulation is, that it is a protective shield that prevents the data from being accessed by the code outside this shield.

In encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of its own class.

A private class can hide its members or methods from the end user, using abstraction to hide implementation details, by combining data hiding and abstraction.

Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

It is more defined with the setter and getter method.

Advantages :

1. Data Hiding

Encapsulation hides the internal details of a class from users.

Users only interact with the class through methods (like getName() or setName()), without knowing how things work inside.

Example: You don't need to know how a car engine works; you just use the key to start it.

2. Increased Flexibility

Encapsulation allows you to control access to the class's variables.

You can decide whether a variable should be:

Read-only: Provide a get method but no set method (e.g., only view someone's name).

Write-only: Provide a set method but no get method (e.g., only set a password but not read it).

This helps enforce rules about how data is accessed and changed.

3. Reusability

Encapsulation makes it easy to reuse code.

Example: Updating how calculations work in a class doesn’t affect other parts of the program that use the class.

5. Freedom to Implement Details

Programmers have full control over how the class's methods and variables are implemented, as long as they provide the promised methods (the interface).

Outsiders (users of the class) don’t care about the internal details—they only interact with the class through the methods.

Example: People use a vending machine to get snacks. They don’t care how the machine processes their request internally, as long as they get the snack.

//This Keyword

1. Using ‘this’ keyword to refer to current class instance variables

// Java code for using 'this' keyword to

// refer current class instance variables

class Test {

int a;

int b;

// Parameterized constructor

Test(int a, int b)

{

this.a = a;

this.b = b;

}

void display()

{

// Displaying value of variables a and b

System.out.println("a = " + a + " b = " + b);

}

public static void main(String[] args)

{

Test object = new Test(10, 20);

object.display();

}

}

Output

a = 10 b = 20

2. Using this() to invoke current class constructor

// Java code for using this() to

// invoke current class constructor

class Test {

int a;

int b;

// Default constructor

Test()

{

this(10, 20);

System.out.println(

"Inside default constructor \n");

}

// Parameterized constructor

Test(int a, int b)

{

this.a = a;

this.b = b;

System.out.println(

"Inside parameterized constructor");

}

public static void main(String[] args)

{

Test object = new Test();

}

}

Output

Inside parameterized constructor

Inside default constructor

4. Using ‘this’ keyword as a method parameter

// Java code for using 'this'

// keyword as method parameter

class Test {

int a;

int b;

// Default constructor

Test()

{

a = 10;

b = 20;

}

// Method that receives 'this' keyword as parameter

void display(Test obj)

{

System.out.println("a = " + obj.a

+ " b = " + obj.b);

}

// Method that returns current class instance

void get() { display(this); }

// main function

public static void main(String[] args)

{

Test object = new Test();

object.get();

}

}

Output

a = 10 b = 20

5. Using ‘this’ keyword to invoke the current class method

// Java code for using this to invoke current

// class method

class Test {

void display()

{

// calling function show()

this.show();

System.out.println("Inside display function");

}

void show()

{

System.out.println("Inside show function");

}

public static void main(String args[])

{

Test t1 = new Test();

t1.display();

}

}

Output

Inside show function

Inside display function

1. What is Enum?

Enum (short for Enumeration) is a special Java class used to define a fixed set of named constants.

Enums provide type safety and make code more readable than using integer or string constants.

2. Characteristics of Enum

✅ Enums are like classes but with predefined constants

✅ Each constant in an enum is an object of the enum type

✅ Enums can have methods, constructors, and instance variables

✅ They can be used in switch statements

Example: Defining an Enum

java

Copy

Edit

enum Status {

RUNNING, FAILED, PENDING, SUCCESS;

}

Using Enum in Java

java

Copy

Edit

Status s = Status.RUNNING;

System.out.println(s); // Output: RUNNING

3. Enum Methods

1️⃣ ordinal() - Returns the index of a constant

java

Copy

Edit

System.out.println(Status.RUNNING.ordinal()); // Output: 0

System.out.println(Status.FAILED.ordinal()); // Output: 1

(Index starts from 0.)

2️⃣ values() - Returns an array of all enum constants

java

Copy

Edit

for (Status s : Status.values()) {

System.out.println(s);

}

Output:

sql

Copy

Edit

RUNNING

FAILED

PENDING

SUCCESS

4. Enum in Conditional Statements

Enums can be used in if-else and switch conditions.

Using if-else

java

Copy

Edit

if (s == Status.RUNNING)

System.out.println("All Good");

else if (s == Status.FAILED)

System.out.println("Try Again");

else if (s == Status.PENDING)

System.out.println("Please Wait");

else

System.out.println("Done");

Using switch

java

Copy

Edit

Status s = Status.PENDING;

switch (s) {

case RUNNING:

System.out.println("All Good");

break;

case FAILED:

System.out.println("Try Again");

break;

case PENDING:

System.out.println("Please Wait");

break;

default:

System.out.println("Done");

break;

}

💡 Advantage: We don’t have to pass the reference variable repeatedly in switch. We only pass constants.

5. Enum Class in Java

✅ All Enums in Java automatically extend java.lang.Enum

✅ Cannot extend another class because Java enums implicitly extend Enum

✅ Every enum constant is an object of the enum type

Example: Printing Enum Superclass

java

Copy

Edit

Status s = Status.SUCCESS;

System.out.println(s.getClass().getSuperclass());

Output:

kotlin

Copy

Edit

class java.lang.Enum

6. Enum with Constructor & Variables

✅ Enums can have constructors, instance variables, and methods

✅ Constructors are executed separately for each enum constant

✅ Instance variables are private, and we use getters to access them

Example: Enum with Constructor

java

Copy

Edit

enum Status {

RUNNING(1), FAILED(2), PENDING(3), SUCCESS(4);

private int code;

Status(int code) { // Constructor

this.code = code;

}

public int getCode() { // Getter method

return code;

}

}

public class EnumExample {

public static void main(String[] args) {

Status s = Status.SUCCESS;

System.out.println("Status: " + s);

System.out.println("Code: " + s.getCode());

}

}

Output:

Enum Class in Java

1️⃣ Key Characteristics of Enums

✔️ Enums cannot extend any other class because they implicitly extend java.lang.Enum.

✔️ Enums can have constructors, methods, and instance variables like regular classes.

✔️ Enums define a fixed set of constants (Enumerated Data Types).

✔️ Each constant in an enum is an object of the enum type.

✔️ Enums provide built-in methods like values(), ordinal(), and name().

Example: Checking the Superclass of an Enum

java

Copy

Edit

enum Status {

SUCCESS, FAILED, PENDING;

}

public class EnumExample {

public static void main(String[] args) {

Status s = Status.SUCCESS;

System.out.println(s.getClass().getSuperclass());

}

}

Output:

kotlin

Copy

Edit

class java.lang.Enum

💡 This proves that all enums extend java.lang.Enum, and they cannot extend any other class.

2️⃣ Enum and Constructors

✔️ Enums can have constructors that are executed when the enum is loaded.

✔️ Each enum constant calls the constructor separately.

✔️ Instance variables in an enum are private, and we use getters to access them.

✔️ If an enum has a parameterized constructor, the default constructor is ignored.

Example: Enum with Constructor

java

Copy

Edit

enum Status {

RUNNING(1), FAILED(2), PENDING(3), SUCCESS(4);

private int code; // Private instance variable

Status(int code) { // Constructor

this.code = code;

}

public int getCode() { // Getter method

return code;

}

}

public class EnumExample {

public static void main(String[] args) {

for (Status s : Status.values()) {

System.out.println(s + " - " + s.getCode());

}

}

}

Output:

sql

Copy

Edit

RUNNING - 1

FAILED - 2

PENDING - 3

SUCCESS - 4

3️⃣ Key Points About Enum Constructors

✔️ The constructor in an enum runs separately for each constant.

✔️ Private constructors ensure that enum objects cannot be created outside the enum.

✔️ We cannot explicitly instantiate an enum using new.

✔️ The number of times the constructor is called = the number of constants in the enum.

Can You Explain This to an Interviewer?

✅ Yes, but in a more structured way as explained above. Your explanation had the correct ideas but needed better clarity. If asked in an interview, you can say:

"Enums in Java are special classes that define a fixed set of constants. They implicitly extend java.lang.Enum, preventing further extension. Enums can have constructors, instance variables, and methods. Each enum constant is an object of the enum type, and its constructor is executed when the enum is loaded."

Annotation in Java :

1)

What is Annotations?

-- annotation is type of comment which is used to provide meta data to the compiler and JVM about the program.

Basic difference between comment and annotations

-- annotation is used to provide meta data to the compiler and JVM about the program but comment is used to provide information to the programmer.

2)

What does annotations provide to program and what does for program?

-- Annotations are used to provide supplemental information about a program.

-- Annotations start with ‘@’.

-- Annotations do not change the action of a compiled program.

-- Annotations help to associate metadata (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.

-- Annotations are not pure comments as they can change the way a program is treated by the compiler. See below code for example.

3)

How annotation prevent us from logical error?

class A{

public void show(){

System.out.println("In A");

}

public void display(){

System.out.println("In A");

}

}

class B extends A{

@Override

public void show(){

System.out.println("In B");

}

/\* @Override

public void displayData(){

System.out.println("In B");

}

\*/

//here you get error because you are not overriding the method of parent class but you are using @Override annotation

// if you are thinking I can override display() method but you should defined new method but whenever you mention @override compiler throw error that you are not override the parent class method

//it helps to avoid the logical error in the program because logical error is more difficult to find when you use annotation it helps to find the logical error

//because solving problem at compile time error is easy than solving problem at runtime

}

#1 what is functional Interface?

--functional interface is an interface with only one abstract method

#2

The properties of a functional interface in Java are as follows:

Single Abstract Method: A functional interface has only one abstract method.

This method is the signature of the interface and defines the behaviour that will be implemented in the lambda expression or with simple class.

Functional Method: The abstract method of a functional interface is called the functional method.

It is the method that provides the functional behaviour of the interface.

@FunctionalInterface Annotation: A functional interface is annotated with the @FunctionalInterface annotation.

This annotation is optional, but it provides a compile-time check that the interface has only one abstract method.

Compatible with Lambda Expressions: A functional interface is compatible with lambda expressions.

A lambda expression is a concise way of implementing the functional method of a functional interface.

(This will discuss in next lecture of lambda expression).

Functional interfaces are a key component of functional programming in Java,

and they are used extensively in the Java 8 Streams API and other Java libraries.

#3 Example

class Main{

public static void main(String []args){

A obj = new B();

obj.show();

//another way to implement functional interface

A obj1 = new A(){

@Override

public void show(){

System.out.println("Hello");

}

};

obj1.show();

}

}

//one way to implement functional interface

class B implements A{

@Override

public void show(){

System.out.println("Hello");

}

}

//specify it is functional interface

@FunctionalInterface //we use this annotation to specify it is functional interface

interface A{

void show();

//void run(); //this will give error as it is a functional interface you cannot write two abstract method in this

}

In this lecture, we are discussing lambda expression:

#1 what is lambda expression in java?

-- In Java, a lambda expression is a concise way to implement a functional interface.

A functional interface is an interface that has only one abstract method, and it can be implemented using a lambda expression.

-- In the previous lecture we see that we are making a class or inner class to implement the functional interface

but using lambda expression we do not need of new class and inner class to implement.

#2 Properties of the lambda expression

Concise syntax: Lambda expressions provide a concise and expressive syntax for defining functional interfaces.

They allow you to define the behaviour of a functional interface in a single line of code.

Functional programming: Lambda expressions are a key component of functional programming in Java.

They allow you to write code that is more declarative and expressive, and less verbose than traditional imperative code.

No need for anonymous inner classes: Lambda expressions provide an alternative to anonymous inner classes, which are often

used to implement functional interfaces in Java. Lambda expressions are more concise and easier to read than anonymous inner classes.

#3 example:

class Main{

public static void main(String []args){

A obj = new B();

obj.show();

//another way to implement functional interface

A obj1 = new A(){

@Override

public void show(){

System.out.println("Hello");

}

};

obj1.show();

//now we can use a lambda expression to implement the functional interface

//you can remove new A() and public void show() and replace it with () -grater then System.out.println("Hello");

//if you have single statements you need not mention curly braces and return keyword

A obj2 = () - grater then System.out.println("Hello");

obj2.show();

//here you can use lambda expression to implement interface C

C obj3= i-grater then System.out.println(i);

obj3.show(5);

}

}

//one way to implement functional interface

class B implements A{

@Override

public void show(){

System.out.println("Hello");

}

}

//specify it is functional interface

@FunctionalInterface //we use this annotation to specify it is functional interface

interface A{

void show();

//void run(); //this will give error as it is not functional interface

}

interface C{

public void show(int i);

}

Lambda Expression with return statements

#1

-- if we have a functional interface

interface A{

int add(int i, int j);

}

in this case, we return int type from add a method for that we can write a lambda expression

in different ways.

-- if we have a single statement you do not need of curly braces

A obj1 =(i,j)- return i+j;

-- if we want to return something in a single statement, no need to mention the return keyword

A obj2 =(i,j)-i+j;

#2

Example:

class Main{

public static void main(String[] args){

// Anonymous class with lambda expression

//if you have single statement and want to return something then you need not to write return keyword and curly braces

A obj= (i,j)- i+j;

System.out.println(obj.add(5,6));

//lambada expression works with functional interface

}

}

interface A{

public int add(int i,int j);

}

#1

types of interface

1) Normal interface

2)functional interface /SAM

3)Marker interface

#2

i)Normal interface

Normal interface contains two or more abstract method.

It can have any number of static and default methods and also even public methods of Object classes.

you can declare variable which are public static final by deault

ii)Functional Interface:

Functional Interface is an interface that has only pure one abstract method.

It can have any number of static and default methods and also even public methods of Object classes

e.g Runnable interface : It contains only run() method

iii)

Marker Interface:

An interface that does not contain any methods, fields, Abstract Methods, and any Constants is Called a Marker interface.

Also, if an interface is empty, then it is known as Marker Interface.

The Serializable and the Cloneable interfaces are examples of Marker interfaces.

/////Exception Handling

Types of Errors in Java

In Java, there are three main types of errors that can occur during the execution of a program:

1️⃣ Compile-Time Errors

🔹 Errors detected during compilation before the program runs.

🔹 Caused by syntax errors, missing semicolons, incorrect variable names, etc.

🔹 The program will not compile if these errors exist.

2️⃣ Logical Errors

🔹 The program runs but produces incorrect output due to a mistake in logic.

🔹 These are hard to detect because the program doesn’t crash.

✅ Example:

int a = 5, b = 10;

System.out.println("Sum: " + (a \* b)); // ❌ Logic error: Should be (a + b), but used multiplication

3️⃣ Runtime Errors (Exceptions)

🔹 Errors that occur during execution and cause the program to terminate.

🔹 Examples include dividing by zero, accessing an invalid array index, and file not found errors.

✅ Example:

int x = 5 / 0; // ❌ ArithmeticException: Division by zero

Exception Handling & Exception Hierarchy

Exceptions in Java are categorized into Checked and Unchecked exceptions.

a) Checked Exceptions

🔹 Checked at compile time → The compiler forces handling using try-catch or throws.

🔹 Mostly related to I/O operations, database access, or network communication.

✅ Example:

import java.io.\*;

class FileExample {

public static void main(String[] args) throws IOException { // Must handle or declare

FileReader file = new FileReader("nonexistent.txt"); // ❌ FileNotFoundException

}

}

b) Unchecked Exceptions

🔹 Not checked at compile time, but can cause crashes at runtime.

🔹 Mostly caused by logical errors, invalid user input, or incorrect API usage.

✅ Examples:

// 1. NullPointerException

String str = null;

System.out.println(str.length()); // ❌ NullPointerException

// 2. ArrayIndexOutOfBoundsException

int[] arr = new int[3];

System.out.println(arr[5]); // ❌ ArrayIndexOutOfBoundsException

clarify that both checked and unchecked exceptions occur at runtime, the key difference is that checked exceptions must be handled at compile time.

Exception Handling in Java (Using try-catch)

Definition:

Exception handling is a mechanism in Java that allows developers to handle runtime errors and ensure that the program does not terminate unexpectedly. It improves program reliability.

Basic Syntax of try-catch:

try {

// Code that may throw an exception

} catch (ExceptionType e) {

// Code to handle the exception

} finally {

// Optional block that always executes

}

Example Code (Handling ArithmeticException):

class Main {

public static void main(String[] args) {

int i = 4; // Regular statement

try {

int a = 10 / 0; // Critical statement (causes ArithmeticException)

} catch (Exception e) {

System.out.println("Some exception occurred: " + e);

}

System.out.println("We are coming out of the try-catch block successfully.");

}

}

Output:

Some exception occurred: java.lang.ArithmeticException: / by zero

We are coming out of the try-catch block successfully.

Types of Statements in Java

Regular Statements :

Execute sequentially without needing special handling.

Examples: Variable declarations, assignments, method calls.

Critical Statements :

Statements that can cause an exception at runtime.

Require special handling using try-catch, throw, or finally.

Example: Division by zero, accessing an invalid index in an array.

Key Takeaways:

✅ If an exception is not handled, JVM terminates the program.

✅ Using a try-catch block prevents abnormal termination and allows the program to continue execution.

✅ The finally block (optional) always executes, whether an exception occurs or not.

Try with Multiple Catch Blocks in Java

In Java, we use a try-catch block to handle exceptions. Sometimes, we may encounter different types of exceptions in the same block of code. To handle them properly, we can use multiple catch blocks.

Example: Handling Multiple Exceptions

int num = 4;

int arr[] = {3, 4, 5};

try {

int result = 40 / num; // May cause ArithmeticException if num = 0

System.out.println(arr[result]); // May cause ArrayIndexOutOfBoundsException

}

catch (ArithmeticException ae) {

System.out.println("Arithmetic Exception: " + ae);

}

catch (ArrayIndexOutOfBoundsException aio) {

System.out.println("Array Index Out of Bounds Exception: " + aio);

}

Explanation:

We are performing division (40 / num), which may cause ArithmeticException (if num = 0).

We are accessing an array index dynamically (arr[result]), which may cause ArrayIndexOutOfBoundsException.

Both statements are inside try because we don't know which will cause an exception.

Handling Parent and Child Exceptions

Java follows exception hierarchy, where some exceptions (like ArithmeticException) are subclasses of more general exceptions (like Exception).

If we catch the parent exception first, it will catch all exceptions, including child exceptions, making the child-specific catch block unreachable.

This causes a compile-time error.

❌ Incorrect Approach (Leads to Compilation Error)

int a = 10;

int arr[] = {3, 4, 5};

try {

int b = 3 / a;

System.out.println(arr[b]);

}

catch (Exception e) { // Parent exception is caught first

System.out.println("Parent class of every exception");

}

catch (ArithmeticException e) { // This will never be reached! ❌

System.out.println("Arithmetic Exception occurred");

}

Error:

error: exception ArithmeticException has already been caught by Exception

✔ Correct Approach (Child Exception First, Then Parent)

int a = 10;

int arr[] = {3, 4, 5};

try {

int b = 3 / a;

System.out.println(arr[b]);

}

catch (ArithmeticException e) { // Child exception first ✅

System.out.println("Arithmetic Exception occurred");

}

catch (Exception e) { // Parent exception after child ✅

System.out.println("Parent class of every exception");

}

Key Takeaways:

✅ Use multiple catch blocks when different exceptions may occur in the same try.

✅ Always catch child exceptions before parent exceptions to avoid compile-time errors.

✅ If the parent exception (Exception) is placed first, the child exception becomes unreachable, causing a compilation error.

Hierarchy of Exceptions in Java

In Java, exceptions are objects that represent errors or unexpected behavior during program execution. Java provides a well-defined exception hierarchy to classify and handle different types of exceptions.

Important Points

✅ All Java classes extend the Object class, so Throwable also extends Object.

✅ The top-level class in the exception hierarchy is Throwable.

✅ The Throwable class has two direct subclasses:

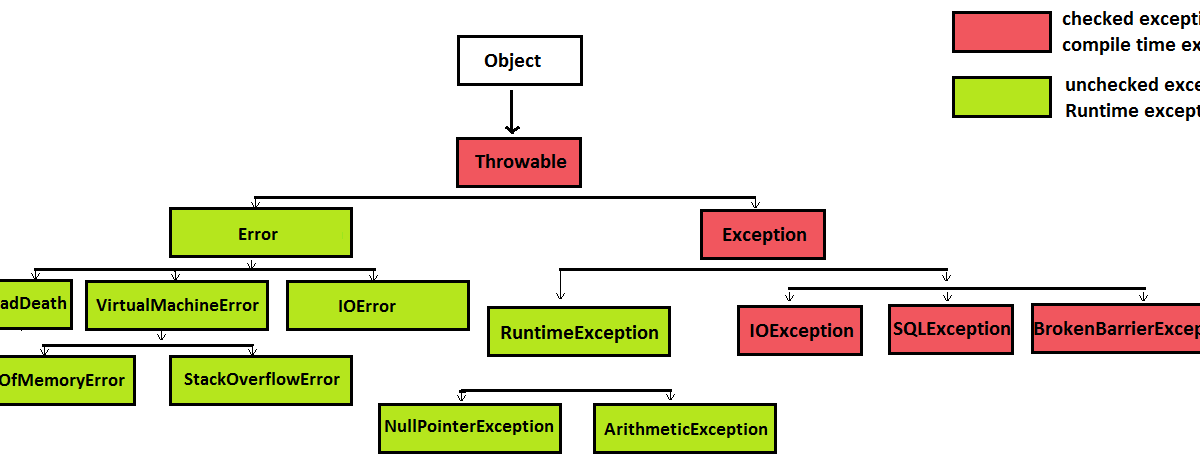
Error → Represents severe issues that cannot be handled (e.g., OutOfMemoryError).

Exception → Represents recoverable errors that can be handled in code.

✅ Checked vs. Unchecked Exceptions:

Checked Exceptions: Must be handled using try-catch or declared using throws (e.g., IOException, SQLException).

Unchecked Exceptions: Do not require handling at compile-time (e.g., NullPointerException, ArithmeticException).



**Throw Keyword in Java**

* The throw keyword is used to explicitly throw an exception in Java.
* When an exception is thrown using throw, the execution of the current method is immediately stopped, and the control is transferred to the nearest matching catch block (if present).
* If no catch block is found, the program terminates with an exception.

**Syntax:**

throw new ExceptionType("Error message");

Example:

public void divide(int a, int b) {

if (b == 0) {

throw new ArithmeticException("Cannot divide by zero");

}

int result = a / b;

System.out.println(result);

}

* In the above example, if b is 0, an ArithmeticException is explicitly thrown, stopping the execution of the method.

**throw vs. try-catch**

* The throw keyword is used **to generate exceptions**.
* The try-catch block is used **to handle exceptions**.
* If an exception is thrown inside a method and not caught within that method, it propagates to the calling method.

**Parameterized vs. Non-Parameterized Exception Constructors**

* Java exceptions have **parameterized constructors** that allow custom messages.
* A non-parameterized constructor creates an exception without a specific message.

**Example with Parameterized Constructor:**

class Main {

public static void main(String[] args) {

int a = 0;

try {

if (a == 0)

throw new ArithmeticException("a should not be zero");

} catch (ArithmeticException e) {

System.out.println("Exception caught: " + e);

}

}

}

**Output:**

less

CopyEdit

Exception caught: java.lang.ArithmeticException: a should not be zero

* The message "a should not be zero" is passed to the ArithmeticException constructor.

**Example: Handling Exceptions with throw and try-catch**

java

CopyEdit

class Main {

public static void main(String[] args) {

int j = 30;

int i = 1;

try {

j = 18 / i;

// Manually throwing an exception based on a condition

if (j == 0) {

throw new ArithmeticException("I do not want to print 0");

}

} catch (ArithmeticException e) { // Catching the exception

System.out.println("ArithmeticException caught");

System.out.println(e.getMessage()); // Using getMessage() to print the custom message

} catch (Exception e) { // Catching any other general exception

System.out.println("Exception caught");

}

}

}

**Key Points:**

* **throw new ArithmeticException("I do not want to print 0");** – This manually throws an exception if j == 0.
* **catch(ArithmeticException e)** – This specifically catches ArithmeticException.
* **e.getMessage()** – Retrieves the custom message passed to the exception.
* **catch(Exception e)** – A general exception handler for any other exceptions.

**1. What is a Custom Exception?**

* Java provides built-in exceptions, but sometimes we need to create **custom exceptions** to handle specific errors in our application.
* A **custom exception** is a user-defined class that extends Exception (checked exception) or RuntimeException (unchecked exception).

**2. How to Create a Custom Exception?**

* We create a custom exception by extending the Exception or RuntimeException class.

**Example:**

java

CopyEdit

class MyException extends Exception {

// Custom exception class

}

* This makes MyException a checked exception.

**3. Adding a Message to a Custom Exception**

* We use a **constructor** to accept an error message and pass it to the parent class using super(message);.

**Example:**

java

CopyEdit

class MyException extends Exception {

public MyException(String message) {

super(message);

}

}

* This allows us to pass a meaningful message when throwing an exception.

**4. Throwing a Custom Exception**

* We use the throw keyword to explicitly throw our custom exception.
* It should be handled using try-catch or declared using throws.

**Example:**

java

CopyEdit

public class Main {

public static void main(String[] args) {

try {

throw new MyException("Custom exception occurred");

} catch (MyException e) {

System.out.println(e.getMessage());

}

}

}

**Output:**

Custom exception occurred

**5. Checked vs. Unchecked Custom Exceptions**

* **Checked Exception**: Extend Exception (must be handled with try-catch or throws).
* **Unchecked Exception**: Extend RuntimeException (optional handling, propagates automatically).

**Example of Unchecked Exception:**

class MyUncheckedException extends RuntimeException {

public MyUncheckedException(String message) {

super(message);

}

}

public class Main {

public static void main(String[] args) {

throw new MyUncheckedException("This is an unchecked exception");

}

}

**6. Why Not Extend Throwable Directly?**

* Throwable is the **superclass of Exception and Error**.
* **Not recommended** because:
  + It is meant for **low-level JVM errors**.
  + Java's built-in exception-handling expects Exception or RuntimeException.

✅ **Recommended:** Extend Exception (checked) or RuntimeException (unchecked).  
🚫 **Not Recommended:** Extending Throwable.

**7. Key Takeaways**

✅ Use Exception for **checked exceptions** (must be handled).  
✅ Use RuntimeException for **unchecked exceptions** (optional handling).  
✅ **Pass error messages** using a constructor and super(message);.  
✅ **Do not extend Throwable** directly.